Exploring Branch Predictors for Constructing Transient Execution Trojans

Tao Zhang
tzhang06@email.wm.edu
William & Mary
Williamsburg, Virginia

Kenneth Koltermann
khkoltermann@email.wm.edu
William & Mary
Williamsburg, Virginia

Dmitry Evtyushkin
devtyushkin@wm.edu
William & Mary
Williamsburg, Virginia

Abstract

Transient execution is one of the most critical features used in CPUs to achieve high performance. Recent Spectre attacks demonstrated how this feature can be manipulated to force applications to reveal sensitive data. The industry quickly responded with a series of software and hardware mitigations among which microcode patches are the most prevalent and trusted. In this paper, we argue that currently deployed protections still leave room for constructing attacks. We do so by presenting transient trojans, software modules that conceal their malicious activity within transient execution mode. They appear completely benign, pass static and dynamic analysis checks, but reveal sensitive data when triggered. To construct these trojans, we perform a detailed analysis of the attack surface currently present in today's systems with respect to the recommended mitigation techniques. We reverse engineer branch predictors in several recent x86_64 processors which allows us to uncover previously unknown exploitation techniques. Using these techniques, we construct three types of transient trojans and demonstrate their stealthiness and practicality.

CCS Concepts • **Security and privacy → Malicious design modifications**; **Side-channel analysis and countermeasures**; **Hardware reverse engineering**;

Keywords Microarchitecture security; reverse-engineering; branch predictor; Trojan; side channel; covert channel; Spectre attack

ACM Reference Format:
Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378526

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378526

1 Introduction

Increased performance of modern processors largely relies on various hardware units performing activities ahead of time. For example, when the processor encounters a branch instruction, a type of instruction that alters the normal sequential execution flow, the branch prediction unit (BPU) predicts the address of the following instruction instead of waiting for the correct address to be computed. In order to avoid damaging the architectural state, execution based on predicted data is performed in a special *transient* (or speculative) mode, which permits roll-backs to previous states. If the prediction is correct, the execution along the predicted path continues. Otherwise, the CPU reverts any changes made by executing incorrect instructions. Recent transient (or speculative) execution attacks, including Meltdown [49] and Spectre [47], demonstrated how such performance optimizations can be manipulated to force victim programs to leak sensitive data by leaving detectable traces in microarchitectural data structures such as CPU caches. These attacks are capable of violating the most fundamental principles of memory safety, including user-kernel isolation. From early 2018, these attacks opened up a new class of microarchitectural threats and quickly spawned many variations [15, 25, 45, 48, 50, 61, 69].

Numerous mitigation techniques have been proposed to protect from transient execution attacks. These techniques range from serializing instructions [4, 39, 46], avoiding dangerous code sequences [5], flushing hardware data structures [2, 4], and limiting transient execution [41, 72] to disabling microarchitectural covert channels [14, 19, 26, 44]. We provide a more detailed description of current protection schemes in Section 2.2. Hardware manufacturers, including Intel and AMD, responded to the threat of transient execution attacks with a series of microcode updates. While being effective in mitigating the main problem, such microcode-based countermeasures noticeably reduce performance [55].

In this paper, we argue against the widely spread perception that the triggers and effects of transient execution attacks are fully understood, and recommended protections

leave no room for any attack to occur. We do so by constructing transient trojans. These malicious software modules conceal their malign functionality in transient execution mode, and unlike previously demonstrated attacks [17, 45, 47, 69], do not require an external attacker controlled process to activate the hidden functionality. First, we perform a reverse engineering study of branch predictor mechanisms in recent Intel and AMD processors and discover several new branch collision triggering techniques. These techniques enable portable, self-contained trojans that can be included in sensitive software (for instance, by a malicious open-source project contributor). Then, we construct software modules that encapsulate all attack components (poisoning and victim branches) inside a single process. Malicious functionality concealed in transient execution mode can remain unnoticed in software even after undergoing rigorous security checks such as symbolic execution [13], taint analysis [20], model checking [27], various methods to detect traditional software backdoors [59, 60, 63, 66, 71], and even existing Spectre detection tools [3, 5, 33, 70]. According to recently proposed transient attack classification by Canella et al. [15], transient trojans described in this paper present a practical example of the same address space transient execution attacks. We argue that transient execution ubiquitously present in nearly all today's CPUs is a natural fit for concealing malicious code since it offers an execution mode that is completely invisible to existing binary and source code analysis techniques.

Paper Contributions In summary, this paper makes the following contributions:

1. We perform a reverse-engineering study[1] of the BPU to uncover the mechanisms responsible for indirect branch prediction and ways to manipulate them. This allows us to construct three types of trojans, each relying on a different BPU anomaly.
2. We present a new branch instruction collision mechanism based on early BPU accesses. First, the mechanism allows attackers to construct trojans that can avoid being detected by current techniques based on code analysis. Second, it permits creations of small and portable trojans.
3. We propose a technique to disperse transient gadgets, improving their stealthiness and effectiveness.
4. We analyze the static prediction mechanism and conclude that it can result in skipped indirect branches, which we use to bypass existing gadget detection techniques and to construct trojans.
5. We present an analysis of current binaries that demonstrates a high prevalence of potentially dangerous collisions reaching hundreds of thousands in large binaries. We argue that such naturally occurring collisions can be used to hide malicious trojans as well as constructing trojans from existing code.
6. Finally, we analyze protection techniques and suggest approaches to remove the threat from uncontrolled transient execution.

[1]Experiments were performed on Intel Haswell (i7-4800MQ), Skylake (i7-6700K), Kaby Lake (i7-8550U), and AMD Ryzen (1950X) machines running recent and fully patched Ubuntu OS with microcode patches installed.

Responsible Disclosure Research findings in this paper have been reported to Intel and AMD.

2 Motivation and Background

2.1 Transient Execution Attacks

Transient execution attacks [8, 15] are based on attacker being able to poison BPU data structures by either executing branch instructions inside an attacker process (Spectre variant 2) or by training BPU structures via supplying specific data (variant 1) [47]. These attacks cause a misprediction by the BPU followed by transient execution of wrong path of instructions. While instructions executing in transient mode cannot modify the architectural state (or write into memory), they can still leave detectable patterns inside microarchitectural structures such as CPU caches. These patterns are not rolled-back after misprediction is detected. A sophisticated attack can be constructed where BPU is poisoned in such a way that CPU first reads sensitive data, then reveals it by leaving detectable traces in microarchiectural structures.

Not all branch mispredictions allow for transient execution attacks. A branch must be unresolved for a number of cycles to allow transient instructions from the wrong execution path to access sensitive data and leave traceable instances by initializing cache accesses. The number of instructions executed in this way, before the branch is resolved, is known as the width of speculative window [33]. Wide speculative windows are created if the information required for the branch resolution is stored in RAM. In this case, a branch can stay unresolved for hundred of cycles [51]. There are two distinct scenarios that create dangerous speculative windows. (1) When the data that determines conditional branch direction (taken or non-taken) is not located in CPU caches, and the BPU mispredicts its direction. (2) When the target of an indirect branch is not in CPU cache while BTB contains an incorrect target due to a collision with another branch. These two scenarios describe Spectre variants 1 and 2 accordingly [47]. The second type (variant 2) of transient execution is potentially more dangerous since it allows the attacker to choose what code will be speculatively executed by poisoning the BTB. Moreover, in such an attack, the attacker can force transient execution to operate in the return-oriented-programming [62] fashion, allowing execution of instructions not present in the original binary [47]. In this paper, we study this type of transient execution attacks.

2.2 Spectre Countermeasures in Existing Systems

Recently, several countermeasures have been developed to mitigate transient execution attacks. The majority of the

proposed techniques focus on mitigating Spectre V2, as it is potentially the most dangerous variation. Although many promising protections techniques have been recently introduced by academia [31, 33, 41, 44, 70, 72], current systems are mostly protected by a few techniques developed by hardware manufacturers and software vendors. Below we summarize a set of protections that are universally enabled on today's systems regardless of OS type. Please note that for simplicity, we focus only on Intel-based machines.

Retpoline Sequences. Spectre v2 attacks require an indirect jump or call instruction to create a wide transient execution window. A simple compile-time solution proposed by Google [67] is to replace all indirect branches with special instruction sequences known as retpolines. These sequences emulate indirect branch functionality by pushing branch targets on stack and then executing a `ret` instruction. When predicting target for returns CPU relies on RSB instead of BTB for which poisoning is significantly more difficult [38]. Although using retpolines is considered an effective countermeasure, recent attacks on the RSB call into question the security of retpoline sequences [48, 50]. In addition, as stated by Intel, Skylake and newer processors are allowed to rely on the BTB for predicting return targets when RSB underflowing occurs [38]. This can make even retpoline-compiled binaries vulnerable.

We performed analysis to find out how common retpolines are on a typical machine. Our analysis included all executables, libraries, and kernel modules on our test machine running the most recent and fully updated version of Ubuntu. We found no retpoline compiled common executables/libraries. The kernel and a small portion of kernel modules were found to be compiled with retpolines resulting in only ≈0.06% of total binaries in the entire system being protected. This is potentially due to developers viewing retpolines as an overkill protection that results in code bloating and performance degradation [55, 64] since the system is already protected with the microcode-based protections.

System-wide Microcode-Based Protections.
Intel responded to transient execution attacks with microcode updates introducing three new features: *indirect branch restricted speculation (IBRS)* which limits speculative execution in privileged modes, *indirect branch prediction barrier (IBPB)*, which prevents cross-process BTB poisoning, and *single thread indirect branch predictors (STIBP)*, which prevent BTB poisoning across hyper-threads [39].

2.3 Current Attack Surface and Motivation

It is important to note that microcode-based protections do not completely eliminate the threat from transient execution. They are designed to protect from known attack scenarios while minimizing performance overhead. For instance, while IBRS by principal is capable to completely disallow speculation of indirect branch targets and thus dangerous transient

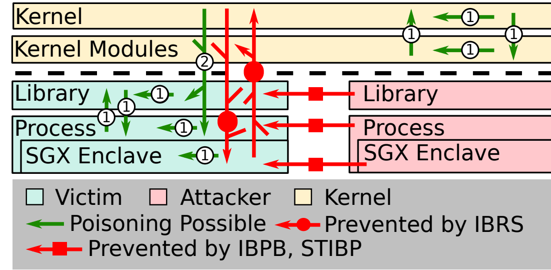


Figure 1. Transient execution attack surface

execution, due to very high performance overhead it is only enabled for kernel, kernel modules, and SGX enclaves on most systems [58]. Similarly, IBPB, together with STIBP, can disallow BTB poisoning between processes and threads, but currently is enforced selectively after performing context switch into a sensitive process [39].

We argue that the currently used protection model still leaves possibilities for attacks. Figure 1 demonstrates a typical attack surface of a fully patched system denoting attack vectors still remaining active. Arrow tail indicates attacker branch, and arrowhead indicates victim branch locations. Two vectors are particularly useful for constructing transient trojans, denoted by ① and ② in the figure. ① is possible because neither IBPB or STIBP can protect against scenarios in which the poisoning branch and the branch being poisoned are located within the same address space. In Section 3.3.1, we demonstrate how such collisions can be easily created by leveraging newly discovered collision patterns. ② is possible because IBRS protects only the code running in privileged modes from being influenced by unprivileged code [2]. This permits kernel to poison the BTB and trigger malicious transient execution inside user process. We explore trojans based on this phenomenon in Section 3.2.

2.4 Threat Model and Assumptions

We assume that the attacker is a malicious developer who is capable of delivering software that seems benign before being activated by a trigger condition. The user (victim) may run static or dynamic analysis and information flow control tools. Moreover, for trojans based on newly discovered collision patterns (Sections 3.1 and 3), the user can run existing Spectre gadget detection tools [3, 5, 33, 70]. The malicious code can be distributed in the form of a precompiled binary, source code, a shared library, or a commit to an open-source project. We assume the attacker has general knowledge about the configuration of the victim's machine, such as CPU microarchitecture generation, versions of shared libraries, and kernel.

[2]IBRS implementation may vary between CPU generations and OS policies enabling or disabling this vector

3 Transient Execution Trojans

In this section, we present transient trojans, programs that can compromise security while containing no malicious instruction sequences in any place reachable by normal execution flow. Even though these trojans appear benign, they output sensitive data when malicious transient execution is activated. The basic building block for a trojan is a condition in which transient execution temporarily violates the architectural state of a program. One of such violations is when two branch instruction collide in BTB. As a result, the body of one branch is executed with data in registers from another branch. This enables a basic memory safety violation, which can lead to sensitive data leakage.

In this section, we describe reverse engineering of mechanisms used to predict indirect branches. We introduce three distinct types of trojans, each utilizing a different kind of BPU anomaly. We show that a malicious developer or an open-source contributor can compose a self-contained software module in which malicious functionality is concealed in transient execution. Unlike previous works [17, 45, 47, 69], which require a separate malicious process controlled by the attacker for BTB poisoning, our self-contained trojans could combine all attack components, including BTB poisoning, in one single process.

3.1 Branch Target Prediction Mechanisms

Modern BPUs are capable of predicting both direct and indirect branches with high accuracy. The mechanisms for predicting targets of these two branch types differ substantially. Figure 2 demonstrates a simplified target prediction mechanism overview. Since the target of a direct branch (including direct calls, jumps, and conditional branches) is fixed, it is predicted by BPU simply caching previously calculated target and storing it in a set-associative BTB [54]. As in any set-associative cache, each lookup is done using *index*, *tag*, and *offset* bits. Index bits determine BTB set for the lookup, while tag and offset allow selection from multiple entries in the same set. To predict the target of a direct branch, the BPU performs a simple lookup based on the branch source address. The address bits are typically hashed to reduce the number of bits stored as tag in BTB.

However, this mechanism is not sufficient for effectively predicting indirect branches because a single indirect branch may jump to different destinations depending upon data the program is processing. Thus a prediction mechanism must account for the context in which the branch is executed. Current BPUs do so by associating indirect branches with patterns of previously executed branches. This is achieved using the mechanism called the branch history buffer (BHB), a shift register structure that serves the purpose of accumulating the branch context. The context is composed by hashing addresses bits of every committed branch instructions with current BHB value [35]. Then compressed BHB value is used to perform target lookups. Such a predictor allows storing multiple targets for a single indirect branch and accurately predicting targets in cases when they depend on previous code sequences.

To maximize the utilization of the BPU storage resources, instead of storing targets for direct and indirect branches in separate structures, both predictors share a single large BTB as in hybrid predictors [16]. The two predictors differ by the type of BTB addressing modes they use: instruction-pointer based (IP-based) and branch history buffer based (BHB-based).

In IP-based addressing, the index, tag, and offset for a BTB lookup are calculated solely based on a subset of the branch instruction virtual address bits. This mode is primarily used for direct branches.

In BHB-based addressing, the lookup is performed based not only on branch instruction address but also on the state of BHB. For instance, compressed BHB value can be used as the BTB tag, allowing to store multiple targets for a single indirect branch. This mode is exclusively utilized by indirect branches. However, when BPU is processing an indirect branch, the two predictors are used concurrently with the prediction selected based on accuracy monitoring for each entry stored in BTB. We provide details further in this section.

While finding an entry based on index calculating and tag matching reminds a normal cache operation, BTB operates differently compared to regular caches. We performed a reverse engineering study to understand the BTB configuration and how branch address bits are used for lookups. We use direct branches to study the IP-based addressing mode. In the first step, we observe that, on Skylake processors, only 30 least significant bits from the branch source address are used for lookups, and the bits [47:30] are ignored, confirming results of previous studies [24]. Then we determine the associativity of the BTB. Assuming bits from the most significant chunk of the remaining [29:0] are used as tag, we create n branch instructions with mismatching tags by flipping these bits. We keep other address bits identical to make matching index and offset. We make each of these branches having a non-matching target. Then we execute this branch sequence twice, observing BTB miss events for any of them during the second time. We use hardware performance counters [1] to detect BPU events. A BTB miss indicates the BTB does not have enough ways in a given set to store all n targets resulting in eviction of one of the targets. We observed no misses for $n < 9$ and a stable miss pattern when $n \geq 9$, indicating that BTB contains 8 ways.

Next, we find which address bits are used as index. To do so, we execute a set of 8 branches that occupies an entire BTB set. Then, one more branch is executed while flipping its address bits in range [29:0]. If the flipped bit is used as tag, all 8+1 branches will have identical indexes and be assigned to the same set. In such case, one of the 8 targets will be evicted.

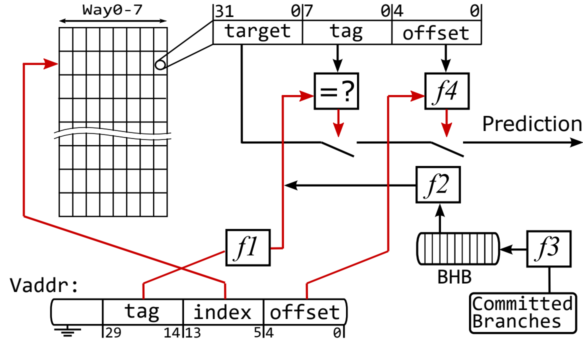


Figure 2. Branch target prediction mechanism combining direct and indirect branch prediction logic. Functions $f1-3$ are bit compression functions; $f4$ is bit matching function. Mechanisms used for trojan construction are highlighted red

However, if the flipped bit is used as index, the branch with the flipped bit will go into a different set, and no evictions will appear. Then we check if any of the 8 branches were evicted from BTB. This way, we identify that bits [13:5] are used as index providing 2^9 total sets resulting in 4 096 total BTB entries. This suggests bits [29:14] used as tag. Previous research [35, 47] determined that tag bits are folded together using a simple XOR operation: `tag=` $a_i \oplus a_{i-8} | i \in [29, 22]$, where a is the branch instruction address. We observed that the exact addressing scheme and bit folding function varies on different microarchitectures. For instance, Haswell processors appear to use `tag=` $a_i \oplus a_{i-9} | i \in [30, 22]$ folding function.

Finally, the remaining bits [4:0] are used as the offset. The exact role of the offset in the context of BTB is unclear. However, the presence of offset is indicated by multiple sources [10, 36, 43]. In general case, the offset can be viewed as a second tag requiring a full match to produce a BTB hit. However, as we discover in Section 3.3.1, the matching is done using a more complex function, which can produce additional collisions resulting in potentially malicious transient execution.

3.1.1 Addressing Modes for Indirect Branch Prediction

Predicting indirect branches based on the context in which they are executed is a logical strategy. Consider a `switch-case` expression in C. It is typically implemented by calculating the resulting target and jumping to this target via an indirect jump instruction. The code pattern executed prior to the `switch` is likely to affect which target will be taken. For this scenario, the BHB-based prediction mechanism is accurate. However, many `switch-case` expressions also have the `default` case, a single target for multiple different (unrecognized) contexts. In this case, the BHB-based mechanism will not be optimal. Instead, the simple IP-based approach will correctly predict the same target regardless of the context.

	Pattern 1					Pattern 2							
Pattern	A	R_1	R_2	R_3	A	A	R_1	B	R_2	A	R_3	B	R_4
Observation	M	M	H	H	H	M	M	M	H	H	H	H	H
Miss rate	0.99	0.99	0.0	0.0	0.05	0.99	0.99	0.99	0.02	0.14	0.0	0.04	0.0

Table 1. Misprediction rate observed in two different patterns composed by varying the BHB context. H represents hit, and M represents misprediction

We hypothesize that BPU uses both mechanisms concurrently.

To verify our reasoning, we designed an experiment in which *the same* indirect branch is executed in multiple different contexts. The contexts are created by varying taken-not-taken patterns of preceding 50 conditional branches. Our experiment included the following contexts: $A \longrightarrow a$, $B \longrightarrow b$ and $R_{1..k} \longrightarrow r$, where $\{A, B, R_{1..k}\}$ are branch contexts and $\{a, b, r\}$ are target addresses for each corresponding context. Contexts A and B have their own targets, while k contexts share a common target r. Executing an indirect branch in different contexts while observing its misprediction rate via hardware performance counters allows us to detect when each addressing scheme is used. For instance, a pattern ABABAB has mispredicted branches for the first two times and correct predictions (hits) for the following ones. This is because the branch predictor quickly learns the dependency between context A and target a and between B and b.

Table 1 presents experimental data collected from running two demonstrative patterns 1 000 times and averaging the results. The first pattern shows how after the branch is executed for the first time, the predictor learns its target to be a. Because of that, it mispredicts the target when we execute it in context $R_1 \longrightarrow r$. However, any consequent execution in random context $R_n \longrightarrow r$ is correctly predicted to go to r. It also shows how the branch is correctly predicted when we execute it in static context $A \longrightarrow a$ second time. These observations show that the branch predictor is capable of predicting the same branch instruction using two independent modes.

The second pattern demonstrates how two addressing modes work in parallel; i.e., the predictor simultaneously checks whether a branch is available using either of the schemes. If it finds a matching tag using any of the schemes, it proceeds with the stored target. In Figure 2, we demonstrated BPU design that can produce such behavior.

These observations allow us to identify two distinct types of indirect branch collisions. Type 1 collisions are when both the BHB state and the reduced branch source address are matched, and the BPU uses BHB-based addressing. Type 2 collisions are when only the branch addresses are matched while mismatching the state of BHB, and the BPU uses IP-based addressing.

3.1.2 Selecting Branch Type for BTB Poisoning

Previous attacks based on BTB poisoning [17, 45, 47, 69] used type 1 collisions. In these works, a victim branch was poisoned from a different process by executing an indirect branch on matching virtual addresses while mirroring the BHB state via repeating behavior of preceding branches. Such setup is less suitable for constructing real-world transient trojans since they must be self-contained; the branch performing poisoning and the branch being poisoned must be located within the same address space. From now on, we refer to the former as **writer branch** or **WB**, and the latter as **reader branch** or **RB**. To construct a trojan based on type 1 collisions, an RB and a WB must be placed at the addresses producing collisions, and have identical BHB states when executing. This is a challenging task due to mapping function $f2$ and BHB update function $f3$ (from Figure 2) unknown or partially reverse-engineered [35, 47]. Even if these functions are fully reverse-engineered, BHB training would require highly irregular code sequences that can be easily detected.

Intuitively, using type 2 collisions is a better option. However, collisions of this type require that both an RB and WB are executed in a new BHB branch context each time. This can be done by running sequences of random taken/not-taken conditional branches before executing WB and RB. This is problematic because unique BHB states will eventually start to repeat, forcing the BPU to switch to the BHB-based mode of addressing. In addition, such code would be highly irregular.

A desired mechanism for constructing trojans must 1) produce reliable collisions when RBs and WBs are located in the same address space; and 2) be easy to mask as benign code. We propose to use direct branches as WBs since 1) they are always handled by the simple IP-based addressing mode making BTB writes more deterministic; and 2) they are common in regular applications with approximately every 4-7$^{\text{th}}$ instruction being a direct branch making them easy to mask as normal code.

3.1.3 Finding Branch Collisions

We hypothesize that the mechanism used to predict direct branches is exactly the same as the IP-based addressing mode for indirect branches. If this hypothesis is true, constructing a trojan becomes straightforward. If we match the address bits used for the tag, index, and offset in a direct WB and an indirect RB, the WB will poison the RB. This will result in speculatively executing code pointed by WB's target when the CPU processes the RB.

To verify this hypothesis, we design an experiment depicted in Figure 3, which allows to reliably identify addresses that result in branch collisions. In this experiment, a direct jump instruction located at address `addrWB` jumping to `addrT1` acts as a WB. An indirect jump is located at `addrRB` jumping to `addrT2` acts as an RB. Then we place a transient gadget at address `addrT3`. This gadget accesses a variable `dat`, loading it into CPU's data cache. If the two branches collide, then mispredicted RB results in transient execution going to `addrT3`, activating the gadget which loads the variable `dat` into the cache. We detect RB mispredictions using hardware performance counters while measuring the latency to access `dat` tells us if the gadget was activated. By moving these branches and gadget instructions in virtual address space and observing collisions, we can effectively scan address space to find addresses that create collisions and analyze corresponding target calculation mechanisms. Using this setup, we make several important observations.

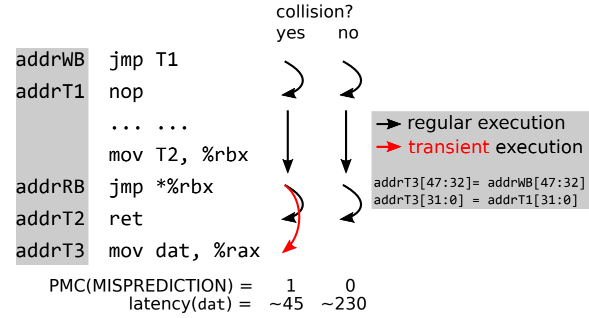


Figure 3. Collision detection experiment setup

Observation 1: Direct branches can serve as WBs, and indirect branches can serve as RBs creating ideal grounds for trojan construction. Moreover, indirect RBs do not need to be executed in a new context every time, as explained in Section 3.1.1.

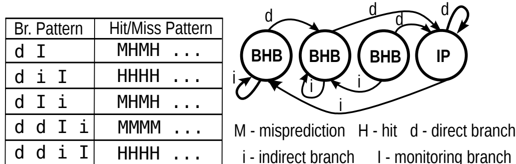
Observation 2: Reduced data stored in BTB (tag and target bits) allows to create collisions within a single process and redirect execution to malicious address. For instance, BTB stores only 32-bit target [47], and to compose the 48-bit prediction target, the CPU simply concatenates branch source address bits [47:32] with the 32-bit target from BTB. This enables attackers to use relative addressing.

Observation 3: We tested different types of branch instructions and concluded that any direct branch can serve as a WB, including calls and conditional jumps.

Observation 4: Our initial tests demonstrated a 50% rate of successful poisoning. However, this rate can be improved if direct a WB is executed multiple times, indicating the possibility of a tournament mechanism [32] selecting the most accurate predictor.

3.1.4 Predictor selector mechanism

To investigate the nature of observation 4, we conduct the following experiment. We place an indirect branch (**i**) and a direct branch (**d**) at colliding addresses and make them having mismatching targets. Since **d** always uses the simple



Br. Pattern	Hit/Miss Pattern
d I	MHMH ...
d i I	HHHH ...
d I i	MHMH ...
d d I i	MMMM ...
d d i I	HHHH ...

Figure 4. Misprediction patterns demonstrating the competition between the two addressing modes and an FSM matching this behavior

IP-based addressing mode, the BTB will contain an incorrect target when predicting **i** using this mode. By preceding **i** with a fixed sequence of conditional branches, we guarantee identical BHB states. As a result, BHB-based mode will always produce a correct prediction. If a tournament mechanism is present, we expect the predictor selector mechanism being affected by executing both of the branches. In particular, when we execute **d**, it will be correctly predicted using the IP-based mode. This will increase the confidence of this mode. In contrast, when **i** is executed, a misprediction from the IP-based mode and a hit from the BHB-based mode will decrease the former and increase the latter predictor's confidence.

Observing **i**'s correct/incorrect prediction patterns allows to detect which predictor is currently in use. A mispredicted observation indicates the IP-based mode usage, while correctly predicted branch tells BHB-based mode is in use. By executing sequences composed from these two branches and observing **i**'s prediction accuracies, we can detect when each predictor wins the tournament. We execute patterns created by invoking **i** and **d** in a random order while collecting the misprediction patterns. Figure 4 demonstrates our observations from several characteristic repeated patterns. Please note that demonstrated prediction patterns are from a single execution of **i** (denoted by capital I) in multiple rounds. By manually inspecting these patterns, we noticed that the observed behavior resembles a finite state machine (FSM) implemented using a 2-bit counter as the system appears switching between 4 stable states. It is possible to manipulate such a mechanism. For instance, executing **i** multiple times in a row increases the accuracy of the BHB-based predictor and makes it more likely to be used for future branches.

In the effort to find the configuration of the FSM responsible for such behavior, we performed the following analysis. First, utilizing the brute-force approach, we generated all possible FSM configurations based on a 2-bit counter. The states of the FSM determine which predictor addressing mode (IP or BHB based) is used. This resulted in 863 040 possible configurations. After removing configurations containing infinite loops and other abnormalities, we reduced this number to 49 104. Next, we simulated these FSMs and ran previously collected patterns through them while observing which predictor is utilized each time. During this stage, we only keep the FSM configurations that match the real system behavior, resulted in only 6 possible unique FSM configurations. We present one such potential FSM in Figure 4. Please note, while this FSM configuration is capable of modeling the real system behavior with high accuracy, the actual mechanism used in the CPU may be different. Knowing the inner workings of the predictor selector, an attacker can perform manipulations forcing the CPU to use the IP-based prediction mode to enable simple collisions by triggering repeated execution of the colliding direct branch instruction.

3.2 Distant Collision Trojans

Now we introduce the first and most basic type of a transient trojan and demonstrate its inner workings. This type of trojan is based on exploiting the BTB addressing scheme where only partial address information is stored. This allows two distinct branches (WB and RB) to collide in a way that when RB is mispredicted, the transient execution goes to the target of WB violating the architectural state. For example, as we described earlier, the tag stored in BTB is folded using a simple XOR operation. Suppose there is a direct branch at address `0x400077` and an indirect branch at `0x4077` in the same process. These branches will collide in BTB when the IP-based addressing mode is used. The attacker can prepare a binary containing branches at colliding addresses. When the binary is deployed on the victim machine, the collision is activated by calling normal API functions in a specific order. In short, this type of transient trojans operates in the following way. First, using a program API, the WB will be activated to write the poisoning entry into the BTB. After that, the attacker trigger conditions for the RB to initialize transient execution, e.g., issuing an API call to access a large array forcing the RB's target to be removed from CPU cache. Then, the RB is executed, and BPU uses the poisoned BTB entry to begin transient execution of a gadget that accesses secret data and reveals secret values using microarchitectural covert channels [9, 18, 23, 29, 30, 37, 42, 52]. We assume the attacker being able to use return-oriented analysis techniques [12, 34, 53] to find or create code sequences (gadgets) that, when executed in transient mode, result in a desired malicious activity. Generally, gadgets can leak data by either 1) leaving traces in shared resources such as CPU data caches [47] or 2) by affecting the timing of certain operations in a controlled way. As demonstrated by Schwarz et al., such delays can be detected over a network [61]. In addition, this type of trojans can be constructed by placing RB and WB in different memory segments within a single application context. For instance, WB can be placed (or existing branch can be utilized) in a library or kernel code segments, while RB being located in trojan's `.text` segment.

Please note, although we construct this type of the trojan utilizing a known branch collision mechanism, we believe that our approach is substantially different. In existing works, a lower privilege entity, such as an untrusted process poisons

a branch in a higher privilege entity such as an OS kernel or an SGX enclave [17, 45, 47, 69]. Such attacks are currently mitigated via IBRS, which protects higher privileged entities (kernel and enclaves) from lower privileged entities. We utilize poisoning vectors that are typically not hindered. In current systems, collisions still occur in many ways as we summarized in Figure 1. The two types of poisoning we will use for constructing trojans are 1) when higher privileged entity poisons a lower privileged entity and 2) when poisoning happens within the same privilege level.

3.2.1 Trojan example utilizing a system call

Assume a malicious developer whose goal is to construct a program that handles secret data and, when triggered, leaks this data. A typical manual inspection or static/dynamic analysis would look for any reference to the sensitive data to make sure they do not reveal it via a covert channel [22]. To show how a practical trojan can be constructed containing no such references, we provide a simple demonstration in which poisoning is triggered by executing a benign existing system call. Performing system calls is a normal activity for any application and unlikely to cause concerns. During a system call, control is temporarily transferred to the operating system. As a result, branches residing in kernel memory trigger writes into the BTB. When the system call is completed, the execution transfers back to the trojan without removing BTB entries placed during the kernel execution. If any of these BTB entries have matching `index`, hashed `tag`, and `offset` bits with an indirect branch in trojan's code, the BPU will treat it as a hit. The predicted address will be composed by concatenating the kernel branch's 32 least significant target bits with the remaining 16 bits from the trojan branch's source address. If such an address contains executable memory, transient execution will take place until CPU detects misprediction and rolls back to the previous state. This will result in violated architectural state. We utilize this phenomenon to construct a trojan that solely relies on normal code executed during a system call to redirect transient execution to a place containing a malicious gadget within trojan's code segment.

During the trojan preparation stage, the developer performs an analysis of the environment in which the future trojan will run and finds a direct branch suitable for poisoning. Typically, this branch needs to be in the final stage of a short system call routine. For our proof-of-concept prototype, we choose a branch inside the `open()` system call. Then the developer introduces a code construction that results in an indirect branch at the colliding address while sensitive data is possible to reference (for instance, the pointer to that data is in one of the registers). This indirect branch transfers regular execution to a benign code containing no leakage instructions. As a result, static analysis will not raise any flags. Modern-day compilers offer a wide range of code constructions that are compiled into code with indirect branches

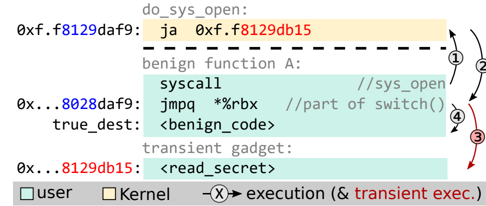


Figure 5. Transient trojan based on `open()` system call

such as virtual functions, function pointers, and computed `gotos`. In addition, a trojan developer can use function alignment and memory mapped code region techniques to easily achieve desired instruction placement.

The next stage of the trojan preparation is finding a suitable transient execution gadget. The gadget must first access the sensitive data and second leak its value via covert channels.

A high level schematic description of the trojan activity is depicted in Figure 5. For each iteration of the attack, the attacker interacts with the trojan via API calls. Each call activates the malicious function inside the trojan, which in turn performs a system call causing BTB poisoning. After the function returns from the system call, it executes an indirect jump, resulting in transient execution of the gadget. After this, the attacker probes the system to obtain microarchitectural traces and recovers leaked data. To evaluate the accuracy of this type of trojan, we collect data from 1 000 rounds of trojan execution. In each round, the gadget is triggered 1 000 times. Then, we count the number of times the gadget is successfully activated. The average success rate for this experiment is 12.79%. Such a rate is within an acceptable range for most microarchitectural attacks. To compare this result to a clean environment, we composed a prototype in which a WB and an RB are both located inside user process memory segments. The average accuracy rate for this configuration is 94.52%. Such a significant improvement is likely due to the normal side effects of system call execution inside the kernel and a mode switch. For instance, system call activity is more likely to evict gadget code from the instruction cache stopping the transient execution attack. Please note that similar trojans can also be constructed by using library functions instead of kernel code. Since library code is placed inside the process address space, IBRS will not prevent the poisoning.

Please note that ASLR and KASLR can make these attacks challenging. However, programs may be compiled without ASLR support and distributed in binary form. Even if KASLR is enabled, its entropy is very small, making attacks still possible by placing RBs at all potential collision addresses. To eliminate the dependency on hardcoded code addresses, we develop two types of portable trojans that work regardless of code placement.

3.3 Portable Trojans

3.3.1 Early Front-end Branch Collisions

Timely branch predictions are very important for the performance of CPU front-end. BPU is responsible for identifying branch instructions early and adjusting fetching to guarantee delivery of instructions from the correct execution path to minimize the number of costly roll-backs. Any slowdown in generating a prediction results in a front-end delay, which propagates into other stages of the pipeline. However, to perform a lookup, BPU needs to know the address of instruction's *last byte*. This is because, typically, BPUs address branches using their least significant byte. On a CISC processor with variable instruction length, such information is not immediately available. A special front-end component, called predecoder, is responsible for detecting instruction boundaries inside a prefetched instruction cache line. We hypothesize that modern-day aggressive front-end designs may avoid waiting for predecode to complete and activate transient execution based only on partial information about potential branch instruction address. This can result in an *early front-end branch collisions* where closely located branches collide due to uncertainty in the boundaries of branch instructions. If this is true, then collisions may appear between branches with mismatching least significant address bits. Several Intel patents [10, 36, 43] refer to these bits as offset while not explaining their exact purpose.

To test the aforementioned hypothesis, we adapted the experiment depicted in Figure 3 with the following changes. First, we position both WB and RB within the same 64 byte instruction cache line. This guarantees matching tag and index bits. Next, we make the direct WB jump to a gadget that now leaks a value stored in register `%rax`. Before executing it, we always load a non-secret value in that register. The indirect RB, as previously, jumps to a benign code. However, prior to that, it loads a secret value into the register `%rax`. If the RB is poisoned by WB, the transient execution shall transfer to WB's body but with secret data loaded in the register. Finally, we execute the WB and RB in a loop and observe effects. If poisoning happens, we detect the secret value leaked via the cache covert channel. An adapted version of this experiment is demonstrated in Figure 8.

We use this setup to scan all possible positions of WB and RB and detect when poisoning happens. As a result, we were able to find stable collision patterns on all tested Intel processors. These patterns indicate a partial offset bits matching mechanism. In particular, on Skylake and Kaby Lake processors: *WB and RB collide either if all offset bits are matched or if bit 5 in WB address is 1 and 0 in RB address.* Thus when generating a prediction for the indirect RB, the BPU mistakingly uses the target of another branch instruction located in one of the subsequent memory locations. On Haswell, a similar pattern exists, however, with bit 4 triggering these

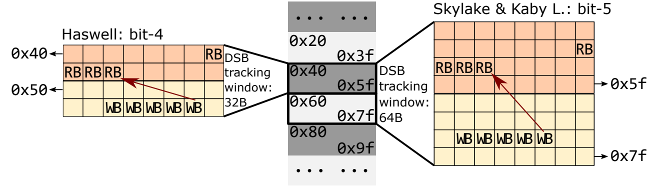


Figure 6. Branch collision patterns within the same cache line on Haswell and Skylake CPUs

collisions instead of 5. These patterns are demonstrated in Figure 6.

This intriguing pattern variation between CPU generations sheds some light on the likely root that causes this collision mechanism. To investigate it, we carefully compared microarchitectural front-end optimizations involved in early instruction processing in Haswell and Skylake processors [6]. Our reasoning is that the mechanism responsible for the behavior must be located in the pipeline before the instruction predecoder and size of instruction blocks it processes is double in Skylake compared to Haswell.

By carefully examining related front-end components [6], we concluded that the decoded streaming buffer (DSB) [7, 57] is a potential root cause. In Intel processors, DSB (also referred as μop cache) helps to avoid decode/predecode delay by storing ready to execute microcode operations (μops). The most performance benefit comes from situations where instruction decoding is delayed, for instance, due to an instruction cache miss or decoders being busy. It also reduces power consumption by suppressing overall decoder activity [65]. Branch prediction while executing μops stored in DSB is equally important for performance as it can trigger μops dispatched directly from the DSB to instruction decode queue, which naturally bypasses all the pre-decoding and decoding stages [56]. However, branch prediction in this stage is challenging due to the specifics of addressing in DSB where the virtual address of only the first instruction inside a tracking window block (32 bytes on Skylake) is stored [40]. Since a single macro instruction can be decoded into a different number of μops; entries in DSB are not aligned with regular instructions in virtual memory. Therefore, the DSB does not have sufficient information on the boundaries of a branch μop. To perform a precise BTB lookup, the DSB logic would have to compute macro-op address from the virtual address of the first μop in the DSB block and the offset. That would significantly increase the mechanism's complexity. Alternatively, DSB can request predictions without specifying the instruction location within its window. We argue that our experimental data suggests the existence of such mechanisms. Our attack example demonstrates how this premature BPU lookup can result in incorrect predictions and malicious transient execution. It is worth mentioning that the size of the DSB tracking window enlarged from 32 Bytes

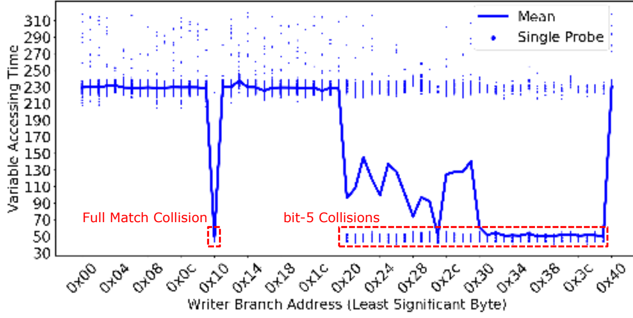


Figure 7. Demonstration of the two collisions types

in Haswell to 64 Bytes in Skylake and Kaby Lake. This may explain the bit-4 and bit-5 observations on these CPUs.

Please note, that this collision mechanism initially appears less stable and is sensitive to surrounding code and branch activity of the program. The average attack success rate in a series of experiments was 4.86%. This is due to this type of collision relying on tight race conditions and contentions inside the front-end components. We tackle this problem by developing an automated collision optimization technique based on an evolutionary algorithm approach in Section 4.

Please note that the collision mechanism described above also works when combined with other collisions types. For example, if two branches have `tag` and `index` bits matched while mismatching higher (ignored) bits ([47:30]), and following the bit-5 collision pattern, the collision will also occur. Figure 7 demonstrates this principle. Presented are results from a Kaby Lake experiment in which we placed an RB at address `0x300110` and then scanned for potential addresses where collisions can occur (`0x100300100` – `0x100300140`) whilst monitoring access time to the variable that is only accessed from transient execution. Low access latency indicates a collision happening. One such collision is between addresses `0x300110` and `0x100300110`. This is due to the full `index`, `tag`, and `offset` match. As seen from the graph, there are additional bit-5 collisions occurring when the WB crosses the 32-byte boundary, and offset collisions start taking place. For simplicity, we will refer to all such collisions as bit-5 collisions regardless of microarchitecture and whether or not they are combined with other collision patterns.

3.3.2 Constructing a Portable Trojan

Trojans based on the early front-end branch collisions can achieve great covertness and portability. This is mainly due to two reasons. First, they do not rely on placing branch instructions far away from each other, contributing to their small size. Second, they do not rely on fixed addresses (aside from offsets within the cache line). This permits them to function when ASLR is enabled. In this type of trojans, all attack components (WB, RB, and the transient gadget) are encapsulated in a small chunk of code that fits into one or few cache lines. As a result, a malicious developer can prepare a portable block of normal C/C++ code that when compiled will act as a trojan. Such trojan will function as expected even if compiler reorders the functions inside binary or the executable is run with ASLR. This opens new vectors for spreading transient trojans. Instead of standalone applications, they can be distributed as shared libraries, patches, or via multi-party software development projects. The requirement for code to be aligned within a 64-byte block is possible to fulfill using various code optimization techniques such as function attributes [28] available in most compilers.

We demonstrate the functionality of the bit-5 collision by creating a simple trojan consisting of two functions, `f1` and `f2`. The high-level overview is presented in Figure 8. We assume `f1` is a function that has access to sensitive data. For instance, this can happen when `f1` is a secret key manipulation function, and the key is loaded in one of the architectural registers in function's prologue (for example `%rax`). In addition, `f1` contains an indirect branch instruction. This can happen because of a `switch()` statement or a call to a virtual method. The code in `f1` does not contain any instructions capable of leaking secret data via covert channels. It assumed that this function will be inspected for that matter. Another function `f2` is a not-sensitive function that is located directly below `f1` in virtual memory, permitting the bit-5 poisoning. Since `f2` does not contain any memory accesses to sensitive data, the presence of a transient gadget in its body does not violate security properties and will not be flagged as dangerous code during analysis. However, due to the branch collision, `f2`'s function body will be executed (in transient mode) in the context of `f1`. By context here we understand the data accessible by each function. This enables a unique transient execution attack. *Due to colliding branches, the architectural state is violated in such a way that results in the body of one function to execute with the context (data) of another function.* For demonstration, we utilize a gadget similar to the gadgets used in prior work [17, 45, 47, 69]. The gadget reads the secret byte and then reveals its value by initiating a memory access using the address dependent on that value.

To evaluate the effectiveness of this type of trojans, we performed an experiment with the code illustrated in Figure 8. We first execute function `f2`, which moves the non-secret value 256 into the register `%rax`. Then it executes the WB, which transfers execution to the gadget that outputs the value stored in the register via leaving a trace in cache. Next, we execute function `f1`, which places the secret value 42 into the same register. Only `f1` has access to that value. The function then activates the RB, resulting in transient execution jumping to the body of `f2`, which contains the gadget leaking the value stored in register `%rax`. Please note, when the gadget instruction is executed in transient mode, the register contains the secret value. After both functions are executed, we probe the cache covert channel by checking all possible byte values transmitted by the gadget (from 0 to

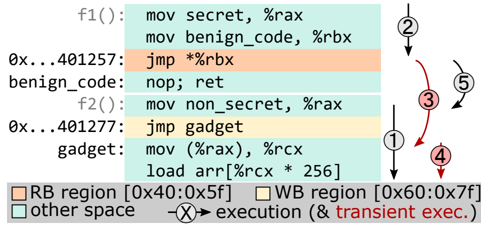


Figure 8. Portable transient trojan example

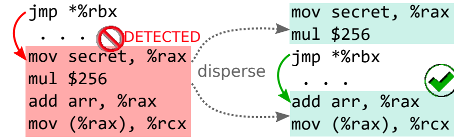


Figure 9. Dispersing a transient gadget to avoid gadget detection tools. Solid arrows indicate transient execution flow

255). If no cache hits are observed, we record no byte transfer. If a transferred value is detected other than 42, we detect an error. Otherwise, we register a correctly transmitted bit. In a real-world trojan, capturing leaked bits is typically performed in another process, or it may affect the timing of an externally observable event. However, for simplicity, we place all components into a single process. In addition, to insure RB's misprediction, we flush the correct target from cache on every iteration. We configured our PoC to leak 10 kilobytes of data and ran it 10 times. The average number of iterations required to transfer 1 byte was 43.69, and the average error rate was 0.0450%. The large number of iterations indicate that bit-5 poisoning does not happen frequently. In Section 4, we present an automated approach to optimizing such trojans allowing to improve their throughput significantly.

3.3.3 Dispersing Gadgets to Avoid Detection

Transient execution attacks rely on gadgets to leak sensitive data. Recently, several works proposed detecting these gadgets [3, 5, 17, 33, 70]. They are largely based on performing static binary analysis. To bypass such detection, we developed a technique based on the newly discovered collision pattern. Static analysis tools rely on detecting code sequences that result in the following actions: 1) memory location is read, and 2) another memory access is performed with an address dependent on the value of the first operation. These solutions use abstract interpretation of binary code to find data dependencies and match activities with known malicious patterns. They are effective in detecting gadgets even if the attacker tries to obfuscate them by using different variables and registers. However, abstract code interpretation does not account for side effects of transient control flow transition due to a bit-5 collision. We can utilize this anomaly to violate the architectural state and disperse a transient gadget into two parts, each of which is not identified as a malicious instruction sequence. Figure 9 shows a gadget consisting of 4 operations. Following the described approach, we add an indirect jump instruction and refactor the code in such a way that the first two operations are executed before the poisoned jump and the last two after. From the architectural state point of view, the second part of the gadget will never be executed. However, due to the poisoning, the transient execution will result in full gadget execution. After this transformation, the code will produce exactly the same transient execution effect. Since we are the first to report the bit-5 collision; we believe that this technique is capable of defeating solutions based on gadget detection.

To evaluate the effectiveness of this technique, we compared the number of iterations required to leak 10KB using the bit-5 based trojan with and without dispersing the gadget. To do that, we moved two of the gadget's instructions before the RB. The average number of iterations required to transfer 1 byte from 10 runs was 20.41, and the average error rate was 0.0147%. These results indicate that dispersed gadgets are roughly two times more efficient. This is due to reducing the number of gadget instructions that execute in transient mode by moving them before the indirect jump. Therefore such a technique can be used not only to avoid detection but also to improve the gadget performance.

3.4 Skipping Branch Trojans

3.4.1 Skipping indirect branches

In addition to collisions between different branches, CPUs we tested based on AMD Ryzen and Intel Haswell architectures have another indirect branch-related anomaly that can be used to construct trojans. In particular, when a prediction is not available in BTB, the CPU simply skips the indirect unconditional branch instruction and proceeds to the following instructions. In addition to constructing trojans, this mechanism can also be utilized to confuse static or dynamic analysis tools. Consider a program in which a certain function is invoked using an indirect call instruction. Assume its target is set during the program initialization and never changes. A detection tool will be able to find this correlation and mark the program safe. However, due to indirect call skipping, a temporal architecture state violation will take place. Intel documentation confirms that indirect branches may be predicted non-taken [6].

3.4.2 Skipping based transient execution attack

The indirect branch skipping mechanism can be utilized to construct trojans with unique properties as they do not rely on known elements of previous Spectre-related attacks. In particular, they do not require conditional branches as in Spectre v1 or branch collisions as in Spectre v2 to violate architectural state.

```
typedef int (*fptr)(void);
int get_sec(){return 42;}
int get_nonsec(){return 0;}

int vuln(){
    int sec, nonsec, tmp;
    fptr f1,f2;
    f1 = get_sec;
    f2 = get_nonsec;
    sec = f1();
    nonsec = f2(); //skipping
    tmp = arr[nonsec * 256];}
```

```
callq   *-0x30(%rbp)
mov     %eax,-0x20(%rbp)          ①
callq   *-0x78(%rbp) //skipping
mov     %eax,-0x24(%rbp)          ②
mov     -0x24(%rbp),%eax
shl     $0x8,%eax
movslq  %eax,%rcx
mov     0x612050(,%rcx,4),%eax
mov     %eax,-0x7c(%rbp)
```

Figure 10. Transient trojan based on branch skipping

To demonstrate the practicality of this approach, we designed a simple trojan application based on this mechanism and complied it using llvm. Figure 10 demonstrates its code with the disassembly of the key elements. Two functions are called via function pointers, and such calls are compiled to indirect call instructions. Function pointer `f1` is used to call the function that returns a secret value, which is then loaded into variable `sec`. The function pointed by `f2` loads a non-secret value into `nonsec`. After these two function calls, a gadget code sequence reveals the value of `nonsec`. Since its value is not secret, it is not considered a violation. According to System V ABI, functions are required to return the values using `%eax` (or `%rax`) register. After the return, caller function stores `%eax`'s value as a local variable on stack.

In the example code, the violation of architectural state happens when function call `f1` is **not** skipped while `f2` is skipped. This results in code ① loading the secret value into register `%eax`, followed by saving it in `sec` and then immediately transmitting execution to code ②, which stores `%eax`'s value in `nonsec`. As a result, both variables temporarily hold exactly the same secret value. Then the gadget successfully reveals the value of the secret data via the cache. Please note that to enable the condition when one function is skipped while another is not, pointers `f1` and `f2` must be located in different cache lines. This can be done by adding or removing local variables in the parent function. For this experiment, we flush `f2` from cache. In a real-world attack, this can be done by finding an eviction set [68].

To evaluate this trojan's accuracy, we executed it on an AMD Ryzen machine leaking 1KB and ran it 10 times. The average number of vulnerable function activations required to leak 1 byte of data was 888.07 with average error rate of 1.74%. Such a relatively low success rate can be explained by the attack relying on an infrequent event when one function is correctly predicted while another is mispredicted. The success rate can be further improved by manipulating with BPU prediction mechanism.

4 Improving Trojan Activation Rate

Effectiveness of transient trojans can be measured by their successful activation rate, which is the percentage of cases when data is leaked compared to total activation attempts. In our initial trojan implementation, the rate appears rather small, for instance, 12.79% and 4.86% for kernel and DSB based trojans, respectively. We noticed that trojans are sensitive to their surrounding code, which can either increase or decrease the success rate. This effect is especially noticeable for portable trojans since they are based on tight race conditions within the CPU front-end. Surrounding code, the code that is executed right before or immediately after the trojan's critical parts can cause various effects (both positive and negative). For instance, it can flush out buffers such as DSB, load store buffer, instruction cache and introduce contention in decoders, functional units, and ports.

Manually tuning trojans for these microarchitecture events is a difficult and meticulous task. First of all, many of the front-end components are not completely reverse engineered. Secondly, fine-tuning one property may affect other properties in a non-trivial way resulting in success rate degradation. Instead of reverse engineering and manual fine-tuning, we propose a method based on genetic programming that enables automatic trojan optimization based on injecting lightweight code artifacts. These artifacts serve no purpose other than creating various microarchitecture conditions and do not affect program's architectural state. Our method is shown to be effective, improving our initial portable trojan implementation from 4.86% to 98.35% resulting in the leakage rate of 13.5 kilobytes per second.

In the first stage of our genetic algorithm approach, we transfer a trojan into a mutation template. This template includes all elements of the original program with additional anchors, places in source code where random activities will be added in the future. The anchors are placed in locations that are likely to interfere with key elements of the trojan, for instance, adjacent to WB and RB. We discovered that trojan accuracy could be affected by adding blocks of `nop` instructions, which affect the code alignment and empty loops that load CPU resources handling branches. For our initial experiment, we used the portable trojan from Section 3.3.2. We placed a total of 15 anchors: 9 `nop` anchors and 6 loop anchors. The `nop` anchors inject 0–150 `nop` instructions while each loop anchor injects a loop with 0–8000 iterations. This results in 10^{43} possible combinations making the brute-force approach not feasible.

Instead, we perform the optimization by starting from 100 initial candidate solutions. We do so by randomly selecting values for each anchor. Then we use a simple genetic algorithm to find an optimal configuration. We set our initial fitness threshold (trojan success rate) at 20%. In each round, we apply an objective fitness function to each candidate, removing all candidates that have an attack rate lower than the fitness threshold. Then we sort the remaining by fitness score. A generator function performing crossover and mutation is applied to a subset of the remaining candidates with the highest fitness scores to create a new variation population of 100 candidates. During this phase, we apply a simple heuristic to avoid crossover between very similar candidates

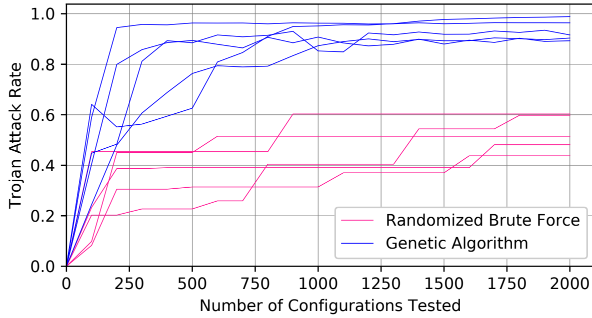


Figure 11. Genetic and randomization optimizer comparison

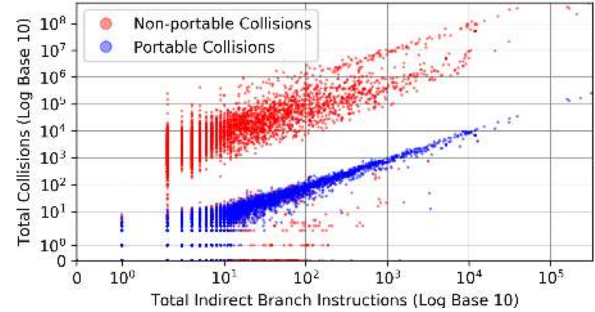


Figure 12. Analysis of branch collisions in existing binaries

ensuring that we continue to have population diversity in each round. This also reduces the risk that our algorithm converges to a suboptimal solution. We also guarantee 20% of each population to be entirely random to increase population diversity.

We compare the genetic programming approach to a simple random-based optimization. Here instead of performing mutation, we keep generating random candidates and select the best performing candidate in each round. Both approaches tested 2 000 trojans in 20 groups, 15 times, and their best 5 runs are demonstrated in Figure 11. The result shows the maximum trojan attack rate only incrementing when a more optimized trojan is found. The genetic algorithm converges to a trojan configurations that produce 90%+ attack rates, finding trojans with high attack rates quicker and 30% higher than the randomization-based approach. That highlights the benefits of using genetic algorithms for optimizing attacks based on microarchitectural effects.

5 Detecting Collisions in Existing Binaries

Branch instruction collisions can occur naturally in regular executables. A typical binary on average contains one direct branch instruction per 4–7 instructions making collisions between indirect and direct branches a common event. An advanced attacker may construct a trojan utilizing these collisions. In this section, we evaluate such naturally occurring collisions in existing binaries and reason about their use in attacks. For our analysis, we use Skylake architecture as a reference. We group all collisions in two types: portable and non-portable. The portable collisions are based on bit-5 mechanism, and their functionality is not tied to hard-coded addresses. Thus they function even in the presence of ASLR, unlike the non-portable collisions, which are based on the distant collision mechanism from Section 3.2. Each executable is analyzed in its normal running context to detect collisions between branches in executable and its libraries.

We developed a light-weight binary analysis tool to find locations where WBs and RBs produce portable and non-portable collisions. First, each binary is disassembled, then we perform a search for all direct and indirect branch instructions. All potential WB and RB instructions are then passed to a BTB mapping function, which is based on Skylake BTB reverse engineering to find their index, tag, and offset bits. Our tool then identifies WB-RB pairs that collide according to two types of collisions.

Figure 12 demonstrates the results gathered from processing 16,015 binaries native to Ubuntu 18.04, including user applications, libraries, and kernel modules. The X-axis shows total indirect branches in executable, while the Y-axis all possible collisions, including collisions between library and code segments. Please note that since distant same address space collisions are sensitive to ASLR, there will be different sets of collisions appearing each time the program is rerun. Although at first this may appear as a negative effect, an advanced attacker can use this phenomenon to further hide a malicious trojan by making it activated only under certain ASLR bits. This makes the analysis of all potential collisions and their effects infeasible. To give a high-level overview of the number of such collisions, we perform the analysis with ASLR deactivated. At the same time, the DSB collisions are not sensitive to ASLR. As seen from the result, existing binaries contain large numbers of naturally occurring collisions of both types. The collisions tend to linearly grow with the total count of indirect branches present in a given binary. For example, Google Chrome executable contains a total of 170k indirect branches resulting in 136k portable and over 300 million non-portable collisions. Such a high number makes hiding malicious branches a relatively easy task as the analysis of all potential transient execution effects becomes very difficult.

As we discussed in Section 3.4.1, indirect branch instructions can violate architectural state even when no collisions are present. Thus, every single indirect call and jump instruction (X-axis in Figure 12) has the potential of doing so. As a result, we believe any indirect branch should be treated as a potential security threat unless CPU design can ensure that transient execution can never leak sensitive data.

6 Countermeasures

Since indirect branch instructions are required for our attacks to function, retpoline sequences can be used as effective mitigation. However, retpolines must be added during compilation and cannot be applied to precompiled binaries. Because retpolines lead to code bloating and performance overhead [11], current binaries seldom use this technique.

Distant same address space branch collisions can be prevented if future BTB designs store full addresses (e.g., tag and target) instead of their reduced or compressed versions. However, such a design would significantly increase the BTB size and, therefore, costs of hardware.

Mitigating bit-5 collisions in hardware appears a more challenging task since it would require a front-end redesign. For instance, a naïve solution is to delay BPU predictions until instruction boundaries are determined. However, that would lead to introducing delays when processing branch intensive μop sequences from DSB. Alternatively, a software-based solution can be developed to sufficiently space direct and indirect branches with binary editing at runtime or by manipulating compiler code generation primitives to prevent placing direct and indirect branches in the same 64-byte block. However, that would lead to significant code bloating. In addition, our collision detection tool can be used to find potentially dangerous branches and inject in-place mitigations such as `lfence` instructions. Future microarchitecture designs are urged to adopt better mechanisms that do not permit branch instruction anomalies, for instance, by adding a type field in the BTB to prevent direct and indirect branch collisions and avoiding indirect branch skipping. A recent work by Yu et al. [73] proposed a light-weight hardware solution based on preventing unsafe data accesses being forwarded to transient execution.

7 Related Work

To the best of our knowledge, this paper is a first work analyzing the security effects of branch collisions within same address spaces. In addition, we introduced a new type of malicious software that utilizes transient execution in the form of self-contained transient trojans represent.

Wampler et al. successfully created a malware program with a transient execution payload [69]. However, malicious software modules presented in their work require a separate activation process. Moreover, a correctly configured IBPB would force BTB flushing on context switched, making poisoning across different processes impossible. All types of our trojans work with current microcode-based protections enabled.

Kiriansky and Waldspurger developed Spectre 1.1, where transient buffer overflows can be used to jump transient execution into arbitrary code. This Spectre buffer overflow attack can be used to redirect execution to instructions after a serialized instruction (Spectre V1 mitigation) [45]. Canella et al. performed an analysis of 12 Spectre variants, including the possibility of multiple same address space Spectre attacks [15]. However, this work did reason on how these vectors can be utilized to construct practical exploits.

Recent works have been published regarding the detection and mitigation of Spectre attacks. SPECTECTOR by Guarnieri et al. detects transient information flows [33], and the principles behind this work can be applicable to the detection of transient trojans. However, without a completely accurate collision model, this and similar tools may overlook dangerous transient execution flows presented in this paper. Our work makes a contribution by expanding upon the existing collision model. Finally, Depoix et al. developed a method of detecting Spectre attacks by identifying Spectre attacks using machine learning [21].

8 Conclusions

In this paper, we presented a new type of practical attack based on transient execution. We demonstrated transient trojans — malicious software modules that utilize BPU anomalies happening inside software entities. In addition, we reverse-engineered the BPU addressing scheme, which allowed us to detect new exploration mechanisms. Utilizing them, we were able to create trojans that have several properties desirable for attackers such as being portable, working in the presence of any microcode-based protection mechanisms, and the ability to stay undetected by current detection tools. We believe our work improves the current understanding of attacks based on transient execution by bridging the gap between exploitable hardware primitives and constructing realistic attacks.

9 Acknowledgments

The work in this paper is partially supported by Intel Corporation and National Science Foundation grant #1850365. The statements made herein are solely the responsibility of the authors and do not necessarily reflect those of the sponsors.

References

[1] 2017. Intel® 64 and IA32 Architectures Performance Monitoring Events. (2017). https://software.intel.com/sites/default/files/managed/8b/6e/335279_performance_monitoring_events_guide.pdf.

[2] 2018. AMD. Software techniques for managing speculation on AMD processors. (2018).

[3] 2018. Detecting Spectre vulnerability exploits with static analysis. (march 2018).

[4] 2018. Intel Analysis of Speculative Execution Side Channels. (2018). https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[5] 2018. LWN.net: Finding Spectre vulnerabilities with smatch. (April 2018). https://lwn.net/Articles/752408/.

[6] 2019. Intel® 64 and IA-32 Architectures Optimization reference Manual. (2019). https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[7] 2019. Wikichip:Skylake(client)-Microarchitectures-Intel. (2019). https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).

[8] Nael Abu-Ghazaleh, Dmitry Ponomarev, and Dmitry Evtyushkin. 2019. How the spectre and meltdown hacks really worked. *IEEE Spectrum* 56, 3 (2019), 42–49.

[9] Murugappan Alagappan, Jeyavijayan Rajendran, Miloš Doroslovački, and Guru Venkataramani. 2017. DFS covert channels on multi-core platforms. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 1–6.

[10] Eran Altshuler, Oded Lempel, Robert Valentine, and Nicolas Kacevas. 2007. Preventing a read of a next sequential chunk in branch prediction of a subject chunk. (Feb. 6 2007). US Patent 7,174,444.

[11] Nadav Amit, Fred Jacobs, and Michael Wei. 2019. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 285–300.

[12] Marco Angelini, Graziano Blasilli, Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, Serena Ferracci, Simone Lenti, and Giuseppe Santucci. 2018. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *2018 IEEE Symposium on Visualization for Cyber Security (VizSec'18)*.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50.

[14] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. 2019. MI6: Secure enclaves in a speculative out-of-order processor. (2019), 42–56.

[15] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. (2019), 249–266.

[16] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. 1996. Branch classification: a new mechanism for improving branch predictor performance. *International Journal of Parallel Programming* 24, 2 (1996), 133–158.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. (June 2019), 142–157. https://doi.org/10.1109/EuroSP.2019.00020

[18] Jie Chen and Guru Venkataramani. 2014. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 216–228.

[19] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B Lee, Haibo Chen, and XiaoFeng Wang. 2018. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 601–608.

[20] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 196–206.

[21] Jonas Depoix and Philipp Altmeyer. 2018. Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning. *Advanced Microkernel Operating Systems* (2018), 75.

[22] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015), 4.

[23] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 843–857.

[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195686

[25] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. 2018. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 693–707.

[26] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. 2018. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1583–1600.

[27] Gordon Fraser, Franz Wotawa, and Paul E Ammann. 2009. Testing with model checkers: a survey. *Software Testing, Verification and Reliability* 19, 3 (2009), 215–261.

[28] GCC. 2020. 6.39 Attribute Syntax. https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html. (2020).

[29] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[30] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. 2016. Your Processor Leaks Information-and There's Nothing You Can Do About It. *arXiv preprint arXiv:1612.04474* (2016).

[31] Abraham Gonzalez, Ben Korpan, Ed Younis, and Jerry Zhao. 2018. Spectrum: Classifying, Replicating and Mitigating Spectre Attacks on a Speculating RISC-V Microarchitecture. (2018).

[32] Michael Gschwind. 2009. Polymorphic branch predictor and method with selectable mode of prediction. (April 21 2009). US Patent 7,523,298.

[33] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2018. SPECTECTOR: Principled Detection of Speculative Information Flows. *arXiv preprint arXiv:1812.08639* (2018).

[34] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. 2012. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 7–7.

[35] Jann Horn. 2018. Reading privileged memory with a side-channel. *Project Zero* (January 2018). https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[36] Bradley D Hoyt, Glenn J Hinton, David B Papworth, Ashwani K Gupta, Michael A Fetterman, Subramanian Natarajan, Sunil Shenoy, and Reynold V D'sa. 1996. Method and apparatus for implementing a set-associative branch target buffer. (Nov. 12 1996). US Patent 5,574,871.

[37] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 639–650.

[38] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. (2018). reference no. 337131-003.

[39] Intel. 2018. Speculative Execution Side Channel Mitigations. (2018). reference no. 336996-003.

[40] David Kanter. 2010. Intel's Sandy Bridge Microarchitecture. https://www.realworldtech.com/sandy-bridge/4/. (2010).

[41] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. (2019), 1–6.

[42] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R Karpuzcu. 2019. POWERT Channels: A Novel Class of Covert Communication Exploiting Power Management Vulnerabilities. In *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture*.

[43] Jonathan Khazam. 2001. Method and apparatus for performing power management by suppressing the speculative execution of instructions within a pipelined microprocessor. (Aug. 28 2001). US Patent 6,282,663.

[44] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.

[45] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *CoRR* abs/1807.03757 (2018). arXiv:1807.03757 http://arxiv.org/abs/1807.03757

[46] Paul Kocher. 2018. Spectre Mitigations in Microsoft's C/C++ Compiler. *Retrieved August* 3 (2018), 2018.

[47] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[48] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[50] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

[51] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. 2018. Let's Not Speculate: Discovering and Analyzing Speculative Execution Attacks. *IBM Technical Report* (2018).

[52] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 46–64.

[53] pakt. 2013. A Turing complete ROP compiler. https://github.com/pakt/ropc. (2013).

[54] Chris H Perleberg and Alan Jay Smith. 1993. Branch target buffer design and optimization. *IEEE transactions on computers* 42, 4 (1993), 396–412.

[55] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, et al. 2018. Measuring the Impact of Spectre and Meltdown. (2018), 1–5.

[56] Lihu Rappoport, Chen Koren, Franck Sala, Ilhyun Kim, Lior Libis, Ron Gabor, and Oded Lempel. 2013. Method and apparatus for pipeline inclusion and instruction restarts in a micro-op cache of a processor. (April 30 2013). US Patent 8,433,850.

[57] Lihu Rappoport, Bob Valentine, Stephan Jourdan, Yoav Almog, Franck Sala, Amir Leibovitz, Ido Ouziel, and Ron Gabor. 2012. Efficient method and apparatus for employing a micro-op cache in a processor. (Jan. 24 2012). US Patent 8,103,831.

[58] Redhat. 2018. Controlling the Performance Impact of Microcode and Security Patches for CVE-2017-5754 CVE-2017-5715 and CVE-2017-5753 using Red Hat Enterprise Linux Tunables. https://access.redhat.com/articles/3311301. (2018).

[59] Elham Salimi and Narges Arastouie. 2011. Backdoor detection system using artificial neural network and genetic algorithm. In *2011 International Conference on Computational and Information Sciences*. IEEE, 817–820.

[60] Felix Schuster and Thorsten Holz. 2013. Towards reducing the attack surface of software backdoors. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 851–862.

[61] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. (2019), 279–299.

[62] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM conference on Computer and communications security*. New York" 552–561.

[63] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.. In *NDSS*.

[64] Nikolay A Simakov, Martins D Innus, Matthew D Jones, Joseph P White, Steven M Gallo, Robert L DeLeon, and Thomas R Furlani. 2018. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. *arXiv preprint arXiv:1801.04329* (2018).

[65] Baruch Solomon, Ronny Ronen, and Doron Orenstien. 2005. Power reduction for processor front-end by caching decoded instructions. (Sept. 27 2005). US Patent 6,950,903.

[66] Sam L Thomas, Flavio D Garcia, and Tom Chothia. 2017. HumIDIFy: a tool for hidden functionality detection in firmware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–300.

[67] Paul Turner and Google Project Zero. 2018. Retpoline: a software construct for preventing branch-target-injection. *Google Help* (2018). https://support.google.com/faqs/answer/7625886

[68] Pepe Vila, Boris Köpf, and José F Morales. 2019. Theory and practice of finding eviction sets. (2019), 39–54.

[69] Jack Wampler, Ian Martiny, and Eric Wustrow. 2019. ExSpectre: Hiding Malware in Speculative Execution. In *26th Annual Network and Distributed System Security Symposium*. NDSS-Symposium, San Diego, CA. https://www.ndss-symposium.org/ndss-paper/exspectre-hiding-malware-in-speculative-execution/

[70] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019).

[71] Chris Wysopal, Chris Eng, and Tyler Shields. 2010. Static detection of application backdoors. *Datenschutz und Datensicherheit-DuD* 34, 3 (2010), 149–155.

[72] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[73] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO âĂŹ52)*. Association for Computing Machinery, New York, NY, USA, 954âĂŞ968. https://doi.org/10.1145/3352460.3358274